Bounding the Effect of Partition Camping in GPU Kernels

Ashwin M. Aji
aaji@cs.vt.edu

Mayank Daga*
mdaga@cs.vt.edu

Wu-chun Feng
feng@cs.vt.edu

Department of Computer Science
Virginia Tech
Blacksburg, Virginia, USA

ABSTRACT

Current GPU tools and performance models provide some common architectural insights that guide the programmers to write optimal code. We challenge and complement these performance models and tools, by modeling and analyzing a lesser known, but very severe performance pitfall, called *Partition Camping*, in NVIDIA GPUs. Partition Camping is caused by memory accesses that are skewed towards a subset of the available memory partitions, which may degrade the performance of GPU kernels by up to *seven-fold*. There is *no* existing tool that can detect the partition camping effect in GPU kernels.

Unlike the traditional performance modeling approaches, we predict a *performance range* that bounds the partition camping effect in the GPU kernel. Our idea of predicting a performance range, instead of the exact performance, is more realistic due to the large performance variations induced by partition camping. We design and develop the prediction model by first characterizing the effects of partition camping with an indigenous suite of micro-benchmarks. We then apply rigorous statistical regression techniques over the micro-benchmark data to predict the performance bounds of real GPU kernels, with and without the partition camping effect. We test the accuracy of our performance model by analyzing three real applications with known memory access patterns and partition camping effects. Our results show that the geometric mean of errors in our performance range prediction model is within 12% of the actual execution times.

We also develop and present a very easy-to-use spreadsheet based tool called *CampProf*, which is a visual front-end to our performance range prediction model and can be used to gain insights into the degree of partition camping in GPU kernels. Lastly, we demonstrate how CampProf can be used to visually monitor the performance improvements in the kernels, as the partition camping effect is being removed.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques

General Terms

Measurement, Performance, Experimentation

Keywords

Partition Camping, Performance Modeling, Micro-benchmarks, Multiple Linear Regression, GPGPU

*Aji and Daga have contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'11, May 3–5, 2011, Ischia, Italy.
Copyright 2011 ACM 978-1-4503-0698-0/11/05 ...$10.00.

1. INTRODUCTION

Graphics processing units (GPUs) are being increasingly adopted by the high-performance computing (HPC) community due to their remarkable performance-price ratio. However, a thorough understanding of the underlying architecture is still needed to optimize the GPU-accelerated applications [18]. Several performance models have recently been developed to study the architecture of the GPU and accurately predict the performance of the GPU-kernels [3, 7, 8, 23]. Our paper both challenges *and* complements the existing performance models by characterizing, modeling and analyzing a lesser known, but extremely severe performance pitfall, called *partition camping* in NVIDIA GPUs.

Partition camping is caused by kernel wide memory accesses that are skewed towards a subset of the available memory partitions, which may severely affect the performance of GPU kernels [16, 17]. Our study shows that the performance can degrade by up to *seven-fold* because of partition camping. Common optimization techniques for NVIDIA GPUs have been widely studied, and many tools and models are available to perform common intra-block optimizations. It is difficult to discover and characterize the effect of partition camping, because the accessed memory addresses and the actual time of memory transactions have to be analyzed together. Therefore, traditional methods that detect similar problems, such as static code analysis techniques to discover shared memory bank conflicts or the approaches used in existing GPU performance models, are prone to errors because they do not analyze the timing information. To the best of our knowledge, we are the first to characterize the partition camping effect in GPU kernels.

In this paper, we deviate from the existing performance models that predict the exact performance of a GPU kernel. Instead, we predict a *performance range* for a given kernel, where its width will indicate the extent to which partition camping can exist in the kernel. The upper and lower bounds indicate the performance levels with and without the partition camping problem respectively. The relative position of the actual kernel performance with respect to the predicted performance range will show the *degree* to which the partition camping problem exists in the kernel. We believe that predicting a performance bound, rather than the exact performance

numbers, is more realistic due to the huge performance variations induced by partition camping.

Next, we discuss our approach to design and develop the performance range prediction model. We first characterize the effects of partition camping by creating a suite of micro-benchmarks, which captures the performance of all the different memory transaction types and sizes, with and without the partition camping behavior. Next, we use the data from the micro-benchmark suite and predict a performance range that bounds the effect of partition camping in real GPU kernels. Our performance prediction model is designed by using rigorous statistical regression procedures. Lastly, we develop and present an extremely user-friendly spreadsheet based tool called *CampProf*, which uses the data from our micro-benchmarks and our performance prediction model and helps the user of the tool to visually detect and analyze the partition camping effects in the GPU kernels. CampProf is a visual front-end to our performance range prediction model to gain insights into the effects of partition camping in GPU kernels. It must be noted that our performance prediction model and the CampProf tool is not meant to replace any of the existing models and tool. They should rather be used in conjunction with the other existing tools, like NVIDIA's proprietary CUDA Visual Profiler (CudaProf) [15] and the CUDA Occupancy Calculator [11], to analyze the overall performance of GPU kernels.

We then perform a detailed experimental analysis on three very different GPU applications with known memory access patterns and partition camping effects, ranging from the molecular modeling domain to graph analysis libraries. We show that our performance prediction model has a geometric mean error of less than 12% when validated against their actual kernel execution times. Next, we demonstrate the utility of the CampProf tool in real applications, and demonstrate how the tool can be used to monitor the performance improvement of the kernel after the partition camping effect has been reduced.

The rest of this paper is organized as follows: Section 2 provides background on the NVIDIA GPU architecture. Section 3 explains our micro-benchmark design to characterize the partition camping problem. Section 4 describes the performance modeling techniques using micro-benchmarks and statistical analysis tools, and the CampProf tool. Section 5 explains the execution characteristics of the chosen applications. Section 6 discusses the experimental results. Section 7 presents the related work. Section 8 concludes the paper and proposes some future work.

2. BACKGROUND ON THE NVIDIA GPUS

In this section, we will describe those aspects of the NVIDIA GPU that are relevant to the discussions in this paper. In this paper, we will restrict our discussions to GPUs with compute capability 1.2 or 1.3, so that the readers will not get distracted by the details of the other architecture families. Section 4.3 describes how our performance model can be easily applied to the other NVIDIA GPU architectures.

The NVIDIA GPU (or *device*) consists of a set of single-instruction, multiple-data (SIMD) streaming multiprocessors (SMs), where each SM consists of eight scalar processor (SP) cores, two special function units and a double precision processing unit with a multi-threaded instruction unit. The actual number of SMs vary depending on the different GPU models. The SMs on the GPU can simultaneously access the *device memory*, which consists of read-write global memory and read-only constant and texture memory modules. Each SM has on-chip memory, which can be accessed by all the SPs within the SM and will be one of the following four types: a set of registers; 'shared memory', which is a software-managed data cache; a read-only constant memory cache; and a read-only texture memory cache. The global memory space is not cached by the device.

CUDA (Compute Unified Device Architecture) [16] or OpenCL (Open Computing Language) [9] are the more commonly used parallel programming models and software environments available to run applications on the NVIDIA GPUs. Massively parallel code can be written via simple extensions to the C programming language. They follow a code off-loading model, i.e. data-parallel, compute-intensive portions of applications running on the host processor are typically off-loaded onto the device. The *kernel* is the portion of the program that is compiled to the instruction set of the device and then off-loaded to the device before execution. The discussions in this paper are not specific to any of the higher level programming abstractions, and can be related to either the CUDA or the OpenCL programming models.

Execution configuration of a kernel: The threads in the kernel are hierarchically ordered as a logical grid of thread blocks, and the CUDA thread scheduler will schedule the blocks for execution on the SMs. When executing a block on the SM, CUDA splits the block into groups of 32-threads called *warps*, where the entire warp executes one common instruction at a time. CUDA schedules blocks (or warps) on the SMs in batches, and not all together, due to register and shared memory resource constraints. The blocks (or warps) in the current scheduled batch are called the *active* blocks (or warps) per SM. The CUDA thread scheduler treats all the active blocks of an SM as a unified set of active warps ready to be scheduled for execution. In this way, CUDA hides the memory access latency of one warp by scheduling another active warp for execution [16, 17]. It follows that the performance of a kernel with an arbitrary number of blocks will be limited by the performance of the set of active blocks (or active warps). So, in this paper, we have chosen 'active warps per SM' as the metric to describe the execution configuration of any kernel, because it is much simpler to be represented in only a single dimension.

There are some hardware restrictions imposed on the NVIDIA GPUs with compute capability 1.2 and 1.3 that limits the possible number of active warps that can be scheduled on each SM. The warp size for the current GPUs is 32 threads. The maximum number of active threads per multiprocessor can be 1024, which means that the maximum number of active warps per SM is 32. Also, the maximum number of threads in a block is 512, and the maximum number of active blocks per multiprocessor is 8 [16]. Due to a combination of these restrictions, the number of active warps per SM can range anywhere from 1 to 16, followed by even-numbered warps from 18 to 32.

Global memory transactions: The global memory access requests by all threads of a half-warp are coalesced into as few memory transactions as possible. The transactions can be from 32-byte, 64-byte or 128-byte segments, based on the size of the word being accessed by the threads. The transaction size will then be reduced, if possible. For example, if all threads in a half-warp access 4-byte words and the transaction size is 64 bytes, and if only the lower or upper half is used, the transaction size is reduced to 32 bytes [16]. The transaction types can either be *read* or *write*, and each of these can have three transaction sizes (32-, 64- or 128-bytes), which means there are *six* possible memory transactions for a kernel running on GPUs with compute capability 1.2 or 1.3.

Note that these global memory transactions can also be triggered in a multitude of other ways, but their performance will not be different. For example, a 32-byte transaction can also be invoked if all the threads in a half warp access the same memory location of any word size. CUDA will compute the least sized transaction for effi-

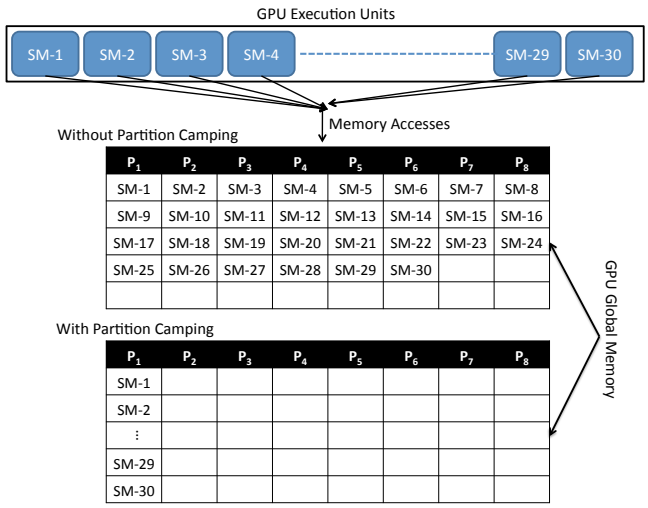


Figure 1: Partition Camping effect in the 200- and 10-series NVIDIA GPUs. Column P_i denotes the i^{th} partition. All memory requests under the same column (partition) are serialized.

ciency purposes. The performance of a global memory transaction does not depend on the method of invocation. We tested this claim by writing a simple micro-kernel, but have not included the details in this paper for brevity.

3. CHARACTERIZING THE EFFECTS OF PARTITION CAMPING

3.1 The Partition Camping Problem

Optimization techniques for NVIDIA GPUs have been widely studied, and many proprietary tools, like CUDA Visual Profiler (CudaProf) and the CUDA Occupancy Calculator spreadsheet tool, guide programmers to perform common intra-block optimizations. These include optimizing arithmetic instruction throughput, coalescing the global memory accesses, and avoiding bank conflicts in shared memory. In this paper, we study a lesser known performance pitfall, which NVIDIA calls 'partition camping', where memory requests across blocks get serialized by fewer memory controllers on the graphics card (Figure 1). Note that each of the above mentioned performance issues can be resolved independent of each other. For example, coalescing can be applied without changing the arithmetic intensity. Similarly, partition camping can be resolved without affecting coalesced accesses or shared memory optimizations, and so on. As shown in the figure 2, there is no existing tool that detects and analyzes the partition camping problem in GPU kernels. Our new tool (CampProf), which we discuss in detail in section 4, aims to detect the partition camping problem in GPU kernels.

Just as shared memory is divided into multiple banks, global memory is divided into either 6 partitions (on 8- and 9-series GPUs) or 8 partitions (on 200- and 10-series GPUs) of 256-byte width. The partition camping problem is similar to shared memory bank conflicts, but experienced at a macro-level where concurrent global memory accesses by all the active warps in the kernel occur at a subset of partitions, causing requests to queue up at some partitions while other partitions go unused [17]. We later show that partition camping can degrade the performance of some kernels by up to *seven-fold*, and so it is important to detect and analyze the effects of this problem.

Tool	GPU Kernel Characteristics					
	Occupancy	Coalesced Accesses (gmem)	Bank Conflicts (smem)	Arithmetic Intensity	Divergent Branches	Partition Camping
CUDA Visual Profiler	✓	✓	✓	✓	✓	✗
CUDA Occupancy Calculator	✓	✗	✗	✗	✗	✗
CampProf	✗	✗	✗	✗	✗	✓

Figure 2: Comparison of CampProf with existing profiling tools. gmem: Global memory; smem: Shared memory.

Discovery of the partition camping problem in GPU kernels is a difficult problem. There is existing literature on static code analysis for detecting bank conflicts in shared memory [4], but the same logic *cannot* be extended to detecting the partition camping problem. Bank conflicts in shared memory occur among threads in a warp, where all the threads share the same clock, and an analysis of the accessed address alone is sufficient to detect conflicts. However, the partition camping problem occurs when multiple active warps queue up behind the same partition *and at the same time*. This means that a static analysis of just the partition number of each memory transaction is not sufficient, and its timing information should also be analyzed. Each SM has its own private clock, which makes the discovery of this problem much more difficult and error prone. Note that the impact of partition camping is severe particularly in memory bound kernels. If the kernel is not memory bound, the effect of memory transactions will not even be significant when compared to the total execution time of the kernel, and we need not worry about the partition camping problem for those cases.

3.2 Designing the Micro-Benchmarks

We develop a suite of micro-benchmarks to study the effect of various memory access patterns combined with the different memory transaction types and sizes. We then show how these micro-benchmarks can characterize the partition camping effect for memory bound kernels. In section 4, we show how the same micro-benchmarks are used to bound the partition camping effect in real applications. Specifically, we predict a range of possible execution times, which denotes the degree to which partition camping can exist in the kernel.

While partition camping truly means that any subset of memory partitions are being accessed concurrently, we choose the extreme cases for our study, i.e. all the available partitions are accessed uniformly (*Without Partition Camping*), or only one memory partition is accessed all the time (*With Partition Camping*). Although this method does not exhaustively test the difference degrees of partition camping, our study acts as a realistic first-order approximation to characterize its effect in GPU kernels. Thus, we developed two sets of benchmarks and analyzed the memory effects with and without partition camping. Each set of benchmarks tested the different memory transaction types (reads and writes) and different memory transaction sizes (32-, 64- and 128-bytes), which made it a total of 12 benchmarks for analysis. As an example, we show in figure 3 that the performance of memory-bound kernels can degrade by up to seven-fold if kernels suffer from partition camping. This particular result was obtained by running a simple 64-byte *memory read* micro-kernel that was part of our micro-benchmark suite, about which we explain next.

Figures 4 and 5 show the kernel of the micro-benchmarks for memory reads, without and with partition camping respectively.

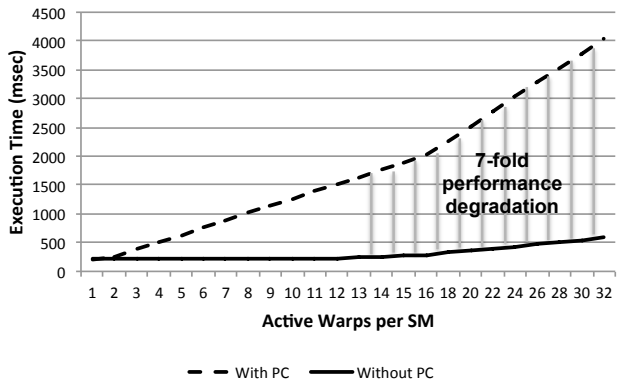


Figure 3: The NegativeEffect of Partition Camping in GPU Kernels. PC: Partition Camping

```
1  // TYPE can be a 2-, 4- or an 8-byte word
2  __global__ void readBenchmark(TYPE *d_arr) {
3      // assign unique partitions to blocks,
4      int numOfPartitions = 8;
5      int curPartition = blockIdx.x % numOfPartitions;
6      int partitionSize = 256; // 256 bytes
7      int elemsInPartition = partitionSize/sizeof(TYPE);
8      // jump to unique partition
9      int startIndex = elemsInPartition
10                              * curPartition;
11     TYPE readVal = 0;
12
13     // Loop counter 'x' ensures coalescing.
14     for(int x = 0; x < ITERATIONS; x += 16) {
15         /* offset guarantees to restrict the
16            index to the same partition */
17         int offset = ((threadIdx.x + x)
18                         % elemsInPartition);
19         int index = startIndex + offset;
20         // Read from global memory location
21         readVal = d_arr[index];
22     }
23     /* Write once to memory to prevent the above
24        code from being optimized out */
25     d_arr[0] = readVal;
26 }
```

Figure 4: Code Snapshot of the Read Micro-benchmark for the NVIDIA 200- and 10-series GPUs (Without Partition Camping). Note: `ITERATIONS` is a fixed and known number.

The benchmarks that simulate the partition camping effect (figure 5) carefully access memory from only a single partition. The micro-benchmarks for memory writes are very similar to the memory reads, except that `readVal` is written to the memory location inside the for-loop (line numbers 21 and 14 in the respective code snapshots). We modify the `TYPE` data-type in the benchmarks to one of 2-, 4- or 8-byte words in order to trigger 32-, 64- or 128-byte memory transactions respectively to the global memory. Although our benchmarks have a high ratio of compute instructions to memory instructions, we prove that they are indeed memory bound, i.e. the memory instructions dominate the overall execution time. We validate this fact by using the methods discussed in [1]. Our suite of benchmarks is therefore a good representation of real memory-bound kernels.

For real memory-bound GPU kernels, we can use their actual number of memory transactions of each type and size, along with

```
1  // TYPE can be a 2-, 4- or an 8-byte word
2  __global__ void readBenchmark(TYPE *d_arr) {
3      int partitionSize = 256; // 256 bytes
4      int elemsInPartition = partitionSize/sizeof(TYPE);
5      TYPE readVal = 0;
6
7      // Loop counter 'x' ensures coalescing.
8      for(int x = 0; x < ITERATIONS; x += 16) {
9      /* all blocks read from a single partition
10        to simulate Partition Camping */
11        int index = ((threadIdx.x + x)
12                        % elemsInPartition);
13        // Read from global memory location
14        readVal = d_arr[index];
15     }
16     /* Write once to memory to prevent the above
17        code from being optimized out */
18     d_arr[0] = readVal;
19 }
```

Figure 5: Code Snapshot of the Read Micro-benchmark for the NVIDIA 200- and 10-series GPUs (With Partition Camping). Note: `ITERATIONS` is a fixed and known number.

our benchmark data, to realistically predict a performance range for any execution configuration of those kernels, as explained in the next section. We can easily obtain the actual number of memory transactions in a kernel by using the CUDA Visual Profiler tool [15] (CudaProf).

4. PERFORMANCE RANGE PREDICTION

In this section, we first design an accurate performance model to predict the range of the effect of partition camping in a GPU kernel. If performance is measured by the wall clock time, the lower bound of our predicted performance will refer to the best case, i.e. without partition camping for any memory transaction. The upper bound will refer to the worst case, i.e. with partition camping for all memory transaction types and sizes. We achieve the accuracy in the model by applying rigorous statistical procedures on the data obtained by running our benchmark suite, which we designed in the previous section.

We then develop and present a very simple easy-to-use tool called *CampProf*, which uses the data from our benchmarks and the performance model, and helps the user of the tool to visually detect and analyze the partition camping effects in the GPU kernel.

Lastly, we discuss how our idea of performance range prediction can very easily be applied to develop similar performance models and tools for the other GPU architectures.

4.1 Performance Model

We perform rigorous statistical analysis techniques to model the impact of partition camping in any memory-bound GPU kernel. We model the effect of memory reads separately from the memory writes, and also model the case with partition camping separately from the case without partition camping. So, we will be designing *four* model equations, one for each of the following cases: (1) Reads, Without partition camping, (2) Writes, Without partition camping, (3) Reads, With partition camping, and (4) Writes, With partition camping. We follow this approach because we believe that modeling at this fine level of detail gives us better accuracy. Specifically, we perform multiple linear regression analysis to fully understand the relationship between the execution time of the different types of our micro-benchmarks and their parameters. The independent variables (predictors) that we chose are: (1) the active warps per SM (w), and (2) the word-lengths that are read or

written per thread. The dependent variable (response) is the execution time (t). The word-length predictor takes only three values (2-, 4- or 8-bytes)[1] corresponding to the three memory transaction sizes, and so we treat it as a group variable (b). This means, we first split the data-type variable into two binary variables (b_2 and b_4), where their co-efficients can be either 0 or 1. If the co-efficient of b_2 is set, it indicates that the word-length is 2-bytes. Likewise, setting the co-efficient of b_4 indicates a 4-byte word-length, and if co-efficients of both b_2 and b_4 are not set, it indicates the 8-byte word-length. We have now identified the performance model parameters, and the performance model can be represented as shown in equation 1, where α_i denotes the contribution of the different predictor variables to our model, and β is the constant intercept.

$$t = \alpha_1 w + \alpha_2 b_2 + \alpha_3 b_4 + \beta \quad (1)$$

Next, we use SAS, a popular statistical analysis tool, to perform multiple linear regression analysis on our model and the data from our benchmarks. The output of SAS will provide the co-efficient values of the performance model.

Significance Test: The output of SAS also shows us the results of some statistical tests, which describe the significance of our model parameters, and how well our chosen model parameters are contributing to the overall model. In particular, R^2[2] ranges from 0.953 to 0.976 and $RMSE$ (Root Mean Square Error) ranges from 0.83 to 5.29. Moreover, we also used parameter selection techniques in SAS to remove any non-significant variable, and choose the best model. This step did not deem any of our variables as insignificant. These results mean that the response variable (execution time) is strongly dependent on the predictor variables (active warps per SM, data-types), and each of the predictors are significantly contributing to the response, which proves the strength of our performance model. Informally speaking, this means that if we know the number of active warps per SM, and the size of the accessed word (corresponding to the memory transaction size), we can accurately and independently predict the execution times for reads and writes, with and without partition camping, by using the corresponding version of equation 1. We then aggregate the predicted execution times for reads and writes without partition camping to generate the lower bound (predicted best case time) for the GPU kernel. Similarly, the predicted execution times for reads and writes with partition camping are added to generate the upper bound (predicted worst case time) for the GPU kernel. We validate the accuracy of our prediction model by analyzing real applications in detail in Section 6.

4.2 The CampProf Tool

4.2.1 *User-Interface Design and Features*

CampProf is an extremely easy-to-use spreadsheet based tool similar to the CUDA Occupancy Calculator [11], and its screenshot is shown in Figure 6. The spreadsheet consists of some input fields on the left and an output chart on the right, which can be analyzed to understand the partition camping effects in the GPU kernel. The inputs to CampProf are the following values: `gld 32b/64b/128b`, `gst 32b/64b/128b`, grid and block sizes, and active warps per SM. These values can easily be obtained from the CudaProf and the CUDA Occupancy Calculator tools. Note that the inputs from just a single kernel execution configuration are enough for CampProf to predict the kernel's performance range for

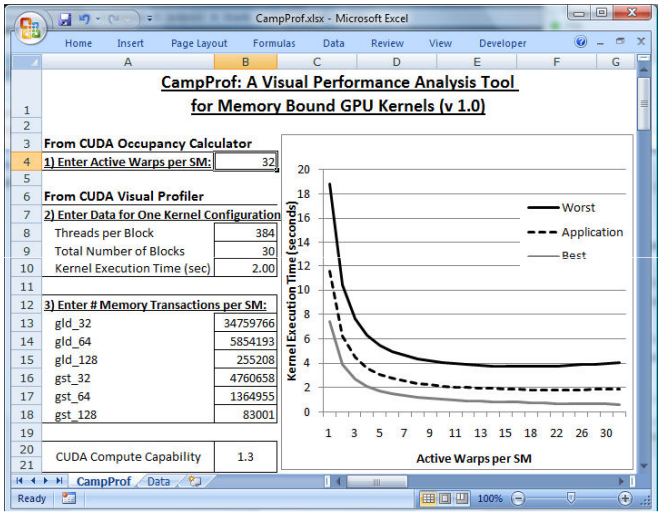


Figure 6: Screenshot of the CampProf Tool.

any other execution configuration. CampProf passes the input values to our performance model, which predicts and generates the upper and lower performance bounds for all the kernel execution configurations[3]. CampProf plots these two sets of predicted execution times as two lines in the output chart of the tool. The best case and the worst case execution times form a band between which the actual execution time lies. In effect, the user provides the inputs for a single kernel configuration, and CampProf displays the execution band for *all* the execution configurations.

In addition, if the actual kernel time for the given execution configuration is provided as input (`GPU Time` counter value from CudaProf), CampProf predicts and plots the kernel execution time at all the other execution configurations, and is denoted by the 'Application' line in the output chart. We predict the application line by simply extrapolating the kernel time from the given execution configuration, in a constant proportion with the execution band. Our performance model is therefore indirectly used to generate this line.

4.2.2 *Visualizing the Effects of Partition Camping*

To detect the partition camping problem in the GPU kernel, the user can simply use CampProf, and inspect the position of the 'Application' line with respect to the upper and lower bounds (execution band) in the output chart. If the application line is almost touching the upper bound, it implies the worst case scenario, where all the memory transactions of the kernel (reads and writes of all sizes) suffer from partition camping. Similarly, the kernel is considered to be optimized with respect to partition camping if the application line is very close to the lower bound, implying the best case scenario. If the application line lies somewhere in the middle of the two lines, it means that performance can be potentially improved, and there is a subset of memory transactions (reads or writes) that is queuing up behind the same partition. The relative position of the application line with respect to the execution band will show the *degree* to which the partition camping problem exists in the kernel. For example, while processing two matrices, the kernel might read one matrix in the row major format (without partition camping) and the other matrix might be read or written into

[1] 1- and 2-byte word lengths will both result in 32-byte global memory transactions.

[2] R^2 is a descriptive statistic for measuring the strength of the dependency of the response variable on the predictors.

[3] As mentioned in Section 2, the 'number of active warps per SM' is our chosen metric of kernel configuration.

in the column major format (with partition camping). This means that only a part of the kernel suffers from camping, and the actual execution time will lie somewhere between the two extremities of CampProf's execution band. Detailed analysis and results will be explained in Section 6. The only remedy to the partition camping problem is careful analysis of the CUDA code and re-mapping the thread blocks to the data, as explained in 'TransposeNew' example of the CUDA SDK [17].

Our approach of predicting a performance range is in contrast to the other existing performance models, which predict just a single kernel execution time. But, our method is more accurate because our model captures the large performance variation due to partition camping.

As previously shown in the figure 2, our performance model and CampProf provide insights into the largely ignored partition camping problem, and *not* the other common performance pitfalls that the CUDA programming guide describes, like non-coalesced global memory accesses, shared memory bank conflicts, low arithmetic intensity, etc. The performance counter values from the CudaProf [15] tool can be used to understand and optimize the common performance bottle-necks of the kernel (Figure 2). However, these values describe the kernel's behavior either within a single SM or a single TPC (depending on the profiler counter), and do not provide any details of the overall system. On the other hand, CampProf helps understand the memory access patterns among all the active warps in the entire kernel. We therefore recommend CampProf to be used along with CudaProf and the CUDA Occupancy Calculator, to detect and analyze all types of performance pitfalls in their GPU kernels.

4.3 Applicability on other GPU Architectures

Our performance model and CampProf can be used to discover the effect of partition camping in any GPU architecture with compute capability 1.3 or lower. For GPU architectures with compute capability less than 1.1 and lower, we will only have to change the code of the micro-benchmarks to include the appropriate number of partitions and the partition size. Then, our present model will still hold and can be directly applied to the new data. The architectural changes in the newer Fermi [13] cards pose new problems for memory-bound kernels. The memory access patterns are significantly different for Fermi, because of the introduction of L1/L2 caches and having only 128-byte memory transactions that occur only at the cache line boundaries. The partition camping problem will still exist in the newer cards, but its effect may be somewhat skewed due to cached memory transactions. Our performance range prediction technique will still work, although in a different scenario. For example, we could bound the performance effect of caching in the GPU kernels, where the upper band indicated the worst case performance of only cache misses, and the lower band indicated the best case performance of only cache hits. This can then help in understanding the effect of improving the locality of memory accesses in the kernel. Our immediate goal is thus to extend CampProf to *CacheProf* for the Fermi architecture.

5. APPLICATION SUITE

In order to validate our performance prediction model, we choose the following three applications with known memory access patterns and partition camping effects: (1) GEM (Gaussian Electrostatic Model) [6], which is a molecular modeling application, (2) Clique-Counter, which is a graph analysis algorithm to count the number of cliques in large bi-directed graphs, and (3) Matrix Transpose. The GPU implementations of GEM [5] and Clique-Counter are part of our own prior and ongoing research, while the Matrix

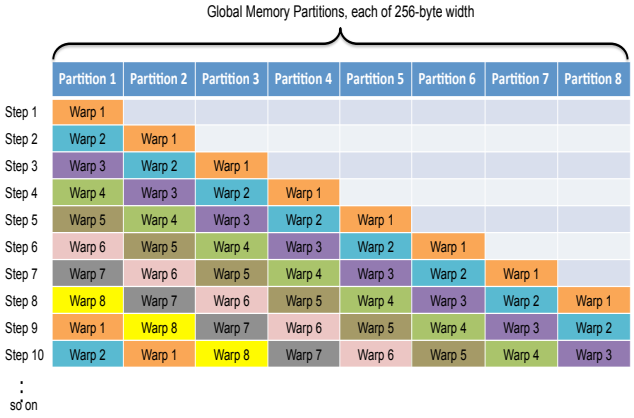


Figure 7: GEM: Memory Access Pattern

Transpose application is part of the NVIDIA CUDA SDK. We will now briefly describe the execution profiles and memory access patterns of these applications, and theoretically analyze the effect of partition camping in them.

5.1 GEM

GEM is a molecular modeling application which allows one to visualize the electrostatic potential along the surface of a macromolecule [6]. GEM belongs to the 'N-Body' class of applications. The goal of this application is to compute the electrostatic potential at all the surface points due to the molecular components. GEM uses an approximation algorithm to speed up the electrostatic calculations by taking advantage of the natural partitioning of the molecules into distant higher level components.

Each GPU thread is assigned to compute the electrostatic potential at one surface point, which is later added together to calculate the total potential for the molecule. The approximation algorithm requires the distance information between each surface point and the higher level components. To compute this distance, each thread needs to access the component coordinates stored in the GPU global memory, which means that GEM can be a memory bound application. Each thread accesses these coordinates in the following order: from the first component to the last, thereby implying that all the active warps would be queued up behind the same memory partition at the beginning of the algorithm. Only one warp can access that global memory partition, which causes the other warps to stall. Once the first warp finishes accessing the elements in the first partition, it would move on the next partition, and the first partition is now free to be accessed by the next warp in the queue. Partition access will thus be pipelined, as shown in Figure 7. Once this memory partition pipeline is filled up (i.e. after eight such iterations on a device with compute capability 1.2 or 1.3), memory accesses will be uniformly distributed across all available memory partitions. It can be assumed that the pipelined nature of memory accesses will not result in further stalls because the workload for all the warps is identical. This illustrates that GEM does not suffer from partition camping.

5.2 Clique-Counter

In graph theory, a clique is a set of vertices in a graph, where every two vertices in the set are connected by an edge of the graph. Cliques are one of the basic concepts of graph theory and also one of the fundamental measures for characterizing different classes of

graphs. We identify the size of a clique by the number of vertices in it. *Clique-Counter* is a program, which as the name suggests, counts the number of cliques of user-specified size in a graph. This is an *NP-complete* problem with respect to the size of the clique that must be counted.

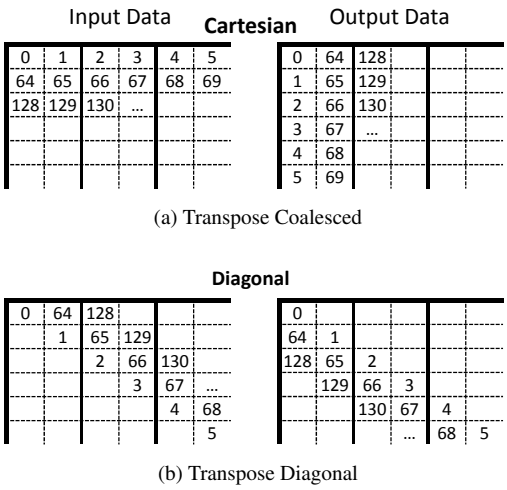
The vertices of the input graph are distributed among GPU threads in a cyclic fashion for load balancing purposes, where the entire graph is stored in the GPU's global memory in the form of adjacency lists. Each thread counts the number of cliques of the given size that can be formed from its set of vertices, followed by a reduction operation that sums up the individual clique counts to get the final result. Larger cliques are formed from smaller cliques by incrementally adding common vertices to the clique. Clique-Counter belongs to the 'backtracking' class of applications, where set intersection operations are repeatedly performed between the vertex set of the current clique and each of their adjacency lists. This means that each thread has to regularly fetch adjacency lists of several vertices from the GPU global memory, which means that the Clique-Counter application can be considered to be memory bound. The memory accesses occur in no particular order and hence, the memory access patterns are neither uniformly distributed across all the memory partitions, nor are they accessing the same partition. Therefore, the Clique-Counter application is neither completely free from partition camping nor is it fully partition camped.

5.3 Matrix Transpose

Matrix Transpose is one of the kernels included in the NVIDIA CUDA SDK. The kernel performs an out-of-place transpose of a matrix of floating point numbers, which denotes that the input and output matrices are stored at different locations in the GPU global memory. The input matrix is divided into square 16×16 tiles, so that the loads are coalesced. Each tile is assigned to one thread-block, which performs the following operations – (i) load the tile from the input matrix (global memory), (ii) re-map the threads to tile elements to avoid uncoalesced memory writes, and (iii) store the tile in the output matrix (global memory). Within-block thread synchronization is required between the above steps to make sure that the global memory writes take place only after all the reads have finished. Since this application predominantly does reads and writes to global memory, it can also be classified as a memory-bound application.

The NVIDIA CUDA SDK provides various versions of matrix transpose, but we specifically chose two of them for our experiments – *Transpose Coalesced* and *Transpose Diagonal*. The only difference between the two applications versions is their global memory access pattern. Figure 8a presents the memory access pattern of Transpose Coalesced. Different solid column groups imply different partitions in the global memory while the numbers denote thread-blocks on the GPU. The figure shows that while reading the input matrix, thread-blocks are evenly distributed among the partitions, however, while writing into the output matrix, all thread-blocks write to the same memory partition. This alludes to the fact that partition camping is not a problem for the read operation but while writing, Transpose Coalesced does suffer from partition camping to a moderate degree. In the Transpose Diagonal version of the application, this problem of partition camping has been rectified by rearranging the mapping of thread-blocks to the matrix elements. The blocks are now diagonally arranged, implying that subsequent blocks are assigned tiles in a diagonal order rather than row-wise. As shown in Figure 8b, blocks *always* access different memory partitions uniformly, thereby, making this version of the application free from partition camping for both reads and writes.

Proof of Memory-Boundedness: Our performance prediction model



(a) Transpose Coalesced

(b) Transpose Diagonal

Figure 8: Matrix Transpose: Memory Access Patterns [17]

can be applied only to memory-bound kernels. This is because if a kernel is not memory bound, the effect of memory transactions and memory access patterns will not form a significant part of the total execution time of the kernel. For such cases, the partition camping problem becomes irrelevant. To rigorously check if our chosen applications are compute-bound or memory-bound, we analyzed the change in their execution times by varying the GPU's core-clock and memory frequencies, and inferred that they all are indeed memory-bound. More details about our experiment can be found in [1]. It must be noted that there are other approaches to categorize the GPU kernels as being compute or memory-bound. We can, say, comment out certain types of instructions in the source code and re-run the kernel to see the effective change in the performance, and then classify the kernel accordingly.

6. RESULTS AND DISCUSSION

In this section, we first explain our experimental setup and then verify our performance prediction model by validating the predicted times against the actual execution times of our chosen applications. Next, we demonstrate the utility of CampProf, which is the front-end to our performance model, for detecting the degree of partition camping in the same applications.

6.1 Experimental Setup

The host machine consists of an E8200 Intel Quad core running at 2.33 GHz with 4 GB DDR2 SDRAM. The operating system on the host is a 64-bit version of Ubuntu 9.04 distribution running the 2.6.28-16 generic Linux kernel. The GPU was programed via the CUDA 3.1 toolkit with the NVIDIA driver version 256.40. We ran our tests on a NVIDIA GTX280 graphics card (GT200 series). The GTX280 has 1024 MB of onboard GDDR3 RAM as global memory. The card has the core-clock frequency of 1296 MHz and memory frequency of 1107 MHz [12]. This card belongs to compute capability 1.3. For the sake of accuracy of results, all the processes which required graphical user interface (GUI) were disabled to limit resource sharing of the GPU.

6.2 Validating the Performance Model

In section 4, we showed how our performance model is used to predict the upper and lower bounds of the GPU kernel performance. In this section, we validate the accuracy of our performance model by comparing the predicted bounds (best and worst case) with the

execution times for those applications with known partition camping behavior. However, we cannot use a single application to validate the accuracy of both the predicted bounds of our model, because an application can exhibit only one type of partition camping behavior. So, we choose different applications from our suite to validate the different predicted bounds of our model. We choose to use geometric mean as the error metric, because it suppresses the effect of outliers and so, is a better estimate for accuracy in the model.

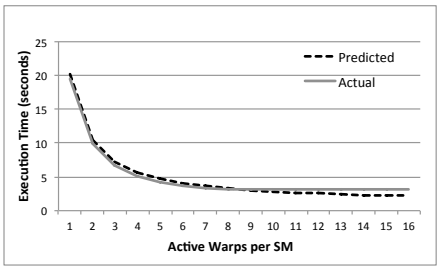
Validating the lower bound (best case): To verify the accuracy of the predicted lower bound, we should compute the error between the actual execution times for an application that is known to be free of partition camping and the predicted best case time by our model for the same application. GEM has been shown to be free of partition camping in the previous section and hence, its execution times are expected to be close to the predicted lower bound of our prediction model.

In Figure 9a, the actual execution times and the predicted lower bound times for GEM have been shown for all the possible configurations. We can see that the predicted best case times ('Best' line from the CampProf output) are agreeing with the actual execution times. The geometric mean of error for the predicted time was found out be 11.7%.

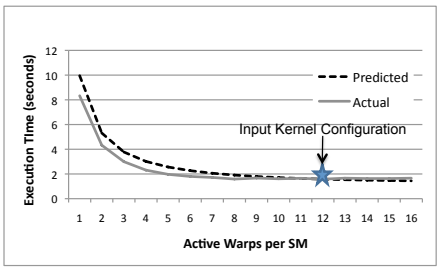
Validating the upper bound (worst case): To verify the accuracy of the predicted upper bound, we should compute the error between the actual execution times for an application that is known to have maximum partition camping effects and the predicted worst case time by our model for the same application. This scenario can only occur if our test application *always* reads and writes to a single partition, so that all the memory transactions of the kernel are serialized. So, validating the predicted upper bound of our model is a non-trivial task for two reasons – (1) It is rare to find applications, other than our micro-benchmarks, which have the maximum partition camping effects, and (2) there is no other available tool or model that detects partition camping, against which we can verify our predicted times. Therefore, it is not possible to directly verify the accuracy of the upper bound of our performance model. We can only use the predicted worst case time as a loose upper bound for the kernel's performance.

Validating the 'Application' line: In section 4.2, we showed that the CampProf tool can be used to predict and plot an 'Application' line in the output chart. This line is predicted by simply extrapolating the kernel time, from the input execution configuration, in a constant ratio to the extremities of the predicted execution band. By validating the 'Application' line, we will indirectly be validating the accuracy of *both* the predicted bounds of our performance model. This means that we can validate the 'Application' line by choosing test applications that are know to have a moderate degree of partition camping, neither the best nor the worst case scenarios. We estimate the accuracy of this extrapolation performed by our model for the Clique-Counter application, because this application is shown to have moderate partition camping effects in section 5.

In Figure 9b, we present the extrapolated 'Application' line (shown as 'Predicted') and the actual times for all possible kernel configurations for the Clique-Counter application, where the starting point for extrapolation is at 12 active warps per SM. But, there are 16 starting points (16 possible execution configurations) from which one can extrapolate to verify the prediction model. To be fair, we chose *all* the 16 execution configurations as starting points for extrapolation for getting the best estimate of our model accuracy. The extrapolated times shown in the figure is for one such starting point (at 12 active warps per SM). The geometric mean of errors due to all such predicted times was found out to be 9.3%.



(a) GEM



(b) Clique-Counter

Figure 9: Validating the Performance Prediction Model

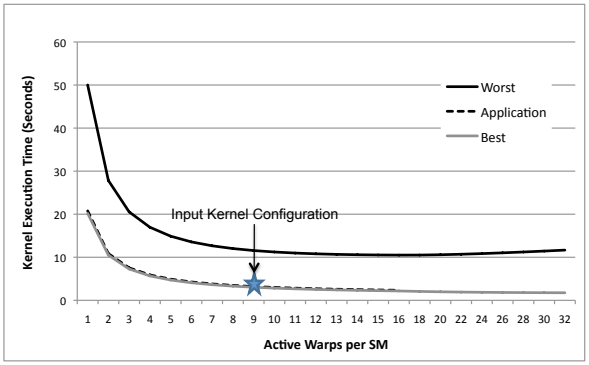
All of the above results indicate that our performance model is accurate in bounding the effect of partition camping in GPU kernels.

6.3 Utility of the CampProf Tool

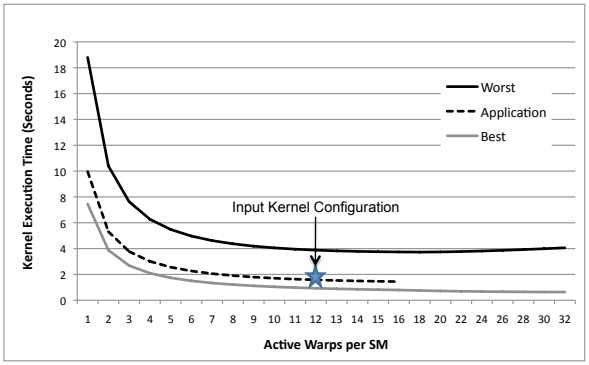
In this section, we demonstrate how the CampProf tool can be used to visually analyze the partition camping effect in our chosen applications. We then use matrix transpose as a case study to show how CampProf is used to monitor the performance of the kernel, where the execution time progresses towards the best case after the partition camping effect has been reduced, i.e. the NVIDIA SDK provides two versions of the transpose example – one with partition camping and another which is supposed to be free from partition camping. The CampProf output can be used to support NVIDIA's claim as well.

Figure 10 shows the CampProf output chart depicting the partition camping effect in all the three applications. The input to the tool is the number of memory transactions and the kernel execution time for one execution configuration (denoted by the $\star$). It shows the worst and best case times for all the execution configurations, as predicted by our model. The 'Application' line in the graphs is extrapolated from the actual execution time that was provided as input. The predicted 'Application' line is not presented for all the execution configurations, i.e. only up to 16 or 24 active warps. This upper limit is calculated by the CUDA Occupancy Calculator, based on the amount of shared memory and registers used by that application.

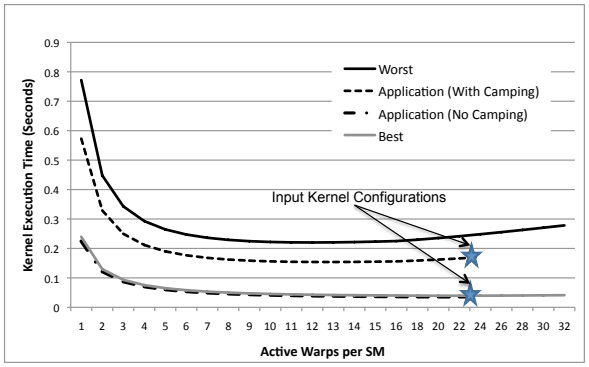
Figure 10a shows that the predicted application performance is extremely close to the 'Best' line of CampProf, which agrees with our discussions about GEM not suffering from partition camping. Similarly, figure 10b indicates that Clique-Counter suffers from a moderate degree of partition camping, because the predicted application line is somewhere in the middle of the predicted execution band. The partition camping effect is not shown to be too severe though. Figure 10c shows the execution band for the Matrix Transpose application, with two predicted 'Application' lines: (1) *'With Camping'* refers to the Transpose Coalesced version of the applica-



(a) GEM



(b) Clique-Counter



(c) Matrix Transpose

Figure 10: CampProf Output: Predicted Performance Ranges

tion, where it is known to partially suffer from partition camping, and (2) *'Without Camping'* refers to the Transpose Diagonal version of the application, where optimizations are performed to get rid of partition camping. CampProf can be thus used to visually demonstrate the performance improvement in a GPU kernel after the partition camping effect has been reduced.

All the predicted 'Application' lines can be used to visually analyze the degree of partition camping in the respective kernels. They also reasonably agree with our claims about the partition camping behavior of the different applications in section 5.

7. RELATED WORK

There have been analytical models developed to help the programmer understand bottlenecks and achieve optimum performance on the GPU. In [3], Baghsorkhi et al. have developed a compiler front end which analyses the kernel source code and translates it into a Program Dependence Graph (PDG) which is useful for performance evaluation. The PDG allows them to identify computationally related operations which are the dominant factors affecting the kernel execution time. With the use of symbolic evaluation, they are able to estimate the effects of branching, coalescing and bank conflicts in the shared memory.

In [7], Hong et al. propose an analytical model which dwells upon the idea that the main bottleneck of any kernel is the latency of memory instructions, and that multiple memory instructions are executed to successfully hide this latency. Hence, calculating the number of parallel memory operations (memory warp parallelism) would enable them to accurately predict performance. Their model relies upon the analysis of the intermediate PTX code generated by the CUDA compiler. However, the PTX is just an intermediate representation which is further optimized to run on the GPU [14]. PTX is not a good representation of the actual machine instructions, and introduces some error in their prediction model.

Recently Ryoo et al. proposed two metrics; efficiency and utilization to prune the optimization space of general purpose applications on the GPU [19]. Their model, however, does not work for memory bound kernels. Boyer et al. present an automated analysis technique to detect race conditions and bank conflicts in a CUDA program. They do so by instrumenting the program to track the memory locations accessed which is done by analyzing the PTX code [4]. Schaa et al. focus on the prediction of execution time for a multi-GPU system, knowing the time for execution on a single GPU [20]. They do so by introducing models for each component of the multi-GPU system; the GPU execution, PCI-Express and the RAM and the Disk.

Micro-benchmarks have been extensively used to reveal the architectural details of the GPUs. In [22], Volkov et al. benchmark the GPUs to tune dense linear algebra. They created detailed benchmarks to reveal the kernel bottlenecks like access patterns of the shared memory and kernel launch overhead. Their benchmarks also characterized the GPU memory sub-system, including the access latencies. Wong et al. also use micro-benchmarks to understand the micro-architecture of the GT200 GPU [23]. Both of them used *decuda*, which is a disassembler for NVIDIA's machine level instructions, to understand the mapping of various instructions on the GPU [21].

In [8], Hong et al. propose an integrated power and performance model for GPUs, where they use the intuition that once an application reaches the optimum memory bandwidth, increasing the number of cores would not help the performance of the application and hence, power can be saved by switching off the additional cores of the GPU. Nagasaka et al. make use of statistical tools like regression analysis and CudaProf counters for power modeling on the GPU [10]. Our work also relies on regression techniques, but we do performance modeling by choosing very different parameters. Bader et al. have developed automated libraries for data rearrangement to explicitly reduce partition camping problem in the kernels [2].

While developing micro-benchmarks and using statistical analysis tools is a common procedure to understand the architectural details of a system, we have used them to create a more realistic performance model than those discussed. We also deviate from the existing literature and predict a *performance range* to understand the extent of partition camping in a GPU kernel. We also provide a simple front-end by developing a spreadsheet like tool, which displays the worst and best possible execution times for any memory bound GPU kernel.

8. CONCLUSIONS AND FUTURE WORK

Key understanding of the GPU architecture is imperative to obtain optimum performance. Current GPU tools and performance models provide some GPU specific architectural insights that guide the programmers to perform common performance optimizations, like coalescing, improving shared memory usage, etc. Our work differs from the existing performance models by characterizing, modeling and analyzing a lesser known, but extremely severe performance pitfall, called partition camping in NVIDIA GPUs. In this paper, we have explored this partition camping problem and have developed a performance model which not only detects the extent to which a memory bound application is partition camped but also predicts the execution times for all kernel configurations if the time for one configuration is known. The performance model was formed using multiple linear regression on the results of our micro-benchmarks which simulate the partition camping effects on the GPU. We have also developed a simple, spreadsheet based tool called CampProf which inherently uses the indigenous model for visual performance analysis.

Our model, at present works only for memory bound applications. In future, we would like to come up with such a performance model for compute bound applications as well. The newer Fermi architecture is known not to suffer from the partition camping problem, however, with its cache hierarchy, it makes GPU programming more challenging. For the Fermi cards, the idea of visual performance analysis can be used to portray the effect of cache misses and to understand the gains of improved data locality. Hence, we would like to develop *'CacheProf'* for the next generation GPU architecture.

Acknowledgments

This work was supported in part by an NVIDIA Professor Partnership Award. We would also like to thank Balaji Subramaniam for helping us with the statistical analysis of our micro-benchmark results.

9. REFERENCES

[1] A. M. Aji, M. Daga, and W. Feng. CampProf: A Visual Performance Analysis Tool for Memory Bound GPU Kernels. Technical report, Virginia Tech, 2010.

[2] M. Bader, H.-J. Bungartz, D. Mudigere, S. Narasimhan, and B. Narayanan. Fast GPGPU Data Rearrangement Kernels using CUDA. In *Proceedings of High Performance Computing Conference*, 2009.

[3] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. mei W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[4] M. Boyer, K. Skadron, and W. Weimer. Automated Dynamic Analysis of CUDA Programs. In *Proceedings of 3rd Workshop on Software Tools for MultiCore Systems*, 2010.

[5] M. Daga, W. Feng, and T. Scogland. Towards Accelerating Molecular Modeling via MultiScale Approximation on a GPU. In *Proceedings of the 1st IEEE International Conference on Computational Advances in Bio and medical Sciences*, 2011.

[6] J. C. Gordon, A. T. Fenley, and A. Onufriev. An Analytical Approach to Computing Biomolecular Electrostatic Potential, II: Validation and Applications. *Journal of Chemical Physics*, 2008.

[7] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th International Symposium of Computer Architecture*, 2009.

[8] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *Proceedings of the 37th International Symposium of Computer Architecture*, 2010.

[9] A. Munshi. The OpenCL Specification, 2008. `http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf`.

[10] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, , and S. Matsuoka. Statistical Power Modeling of GPU Kernels Using Performance Counters. In *Proceedings of International Green Computing Conference*, 2010.

[11] NVIDIA. CUDA Occupancy Calculator, 2008. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[12] NVIDIA. GeForce GTX 280 Specifications, 2008. `http://www.nvidia.com/object/product_geforce_gtx_280_us.html`.

[13] NVIDIA. NVIDIA Fermi Compute Acrhitecture, 2008. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[14] NVIDIA. The CUDA Compiler Driver NVCC, 2008. `http://www.nvidia.com/object/io_1213955090354.html`.

[15] NVIDIA. CUDA Visual Profiler, 2009. `http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/cudaprof_1.2_readme.html`.

[16] NVIDIA. NVIDIA CUDA Programming Guide-2.3, 2009. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`.

[17] NVIDIA. Optimizing Matrix Transpose in CUDA, 2009. `NVIDIA_CUDA_SDK/C/src/transposeNew/doc/MatrixTranspose.pdf`.

[18] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, and W. mei Hwu. Program Optimization Study on a 128-Core GPU. In *Workshop on General Purpose Processing on Graphics Processing*, 2008.

[19] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code Generation and Optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[20] D. Schaa and D. Kaeli. Exploring the Multi-GPU Design Space. In *Proc. of the IEEE International Symposium on Parallel and Distributed Computing*, 2009.

[21] W. van der Laan. Decuda, 2008. `http://wiki.github.com/laanwj/decuda`.

[22] V. Volkov and J. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008.

[23] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *Proceedings of the 37th IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.